Tools For Evaluating The Quality of C Programs

by

Mantri Kamal Kishore

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY KANPUR

MARCH, 1995

Tools For Evaluating The Quality Of C Programs

A Thesis Submitted
in Partial Fulfilment of the Requirements
for the Degree of

Master of Technology

by

Mantri Kamal Kishore

under the guidance of

Dr. Pankaj Jalote

to the

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY KANPUR

March 1995

Certificate

It is certified that the work contained in this thesis entitled **Tools For Evaluating the Quality Of C Programs**, by **Mantri Kamal Kishore (Roll No: 9311113)** under my supervision, and that this work has not been submitted elsewhere for a degree.

Pankaj Jalote

Dr. Pankaj Jalote
Assistant Professor,
Department of Computer Science and Engineering
IIT, Kanpur

March 1995

Acknowledgements

I am greatly indebted to my thesis supervisor *Dr. Pankaj Jalote*, for his consistent support, invaluable suggestions and friendly guidance extended to me throughout the project. He has introduced me to the exciting field of Software Metrics. I also wish to thank him for painfully going through my thesis report and pointing out many corrections.

I thank Ram Kumar for his help in academic matters and for giving nice company throughout my stay at IIT/K. I would also like to thank the class of M. Tech. '93 for making the last one and half years, a memorable period of my life. I also greatly indebted to the Hall V T.V. for giving me countless joyful moments.

Lastly, but not least, I would like to thank to my parents for their constant encouragement and motivation to pursue higher studies.

March 23, 1995 **Kamal Kishore Mantri**

Abstract

Software metrics *are the quantitative measures that can be algorithmically derived from any part of the software life cycle or any product the cycle produces. Many metrics to measure the complexity of software have been proposed in the literature. The different complexity metrics proposed measure the complexity from different perspectives and with different assumptions. In this work, we automated the collection of a few different software complexity metrics, of which some of them can be applied at the end of design stage and some after the coding stage. Based on the complexity measures, tools can also be used to highlight the likely problematic areas of a software system.*

Using the developed tools, we conducted various experiments on three pilot projects to experimentally check if the highlighted modules are indeed more error prone, and whether any correlation exists between these measures and actual error data of the projects. We also performed experiments to study the correlation between different metrics. Our experiments show that there is a fair correlation between some of the metrics, that modules highlighted by some proposed methods indeed are frequently error prone, and that size is an important parameter affecting complexity.

Key-words and phrases : *Software Metrics, Software Measurement, Software Quality Control, Software Complexity*

Contents

List of Figures

CHAPTER 1

Introduction

Over the past several years, there has been a drastic change in hardware and software prices. As the cost of hardware continued to fall, software cost increased steadily. In many applications, cost of software accounts for more than the total amount spent on computing hardware. Fair amount of resources should be spent on communication and coordinating activities, when size of the system is large. Several studies [5] have shown that cost and schedule of a project increases exponentially as the the system size grows. So there is a need to develop large and complex software systems, having high degree of quality within time and cost constraints. *Software Engineering* is a systematic, and scientific approach for achieving this.

During the software project development, software managers have to monitor the progress of the project, and take necessary actions to handle both foreseen and unforeseen events. To take necessary actions, they need proper metrics to quantify the required information, otherwise subjective opinion has to be used, which is unreliable, and contradicts the basic goals of software engineering.

Similarly, some quantifiable measures are required to measure software quality. Software to be developed has to posses various quality attributes like, *reliability, testability, maintainability, robustness, reuseability, portability, userfriendliness, verifiability, etc.* Software quality can not be achieved unless we are able

to specify it properly. Quantitative quality requirements specification should be done at the beginning of software development. Then, these specifications could be demonstrated objectively, at the end of software product construction. Quantifiable quality requirement specifications also provide software managers a rational basis for allocating personnel, and computing resources to quality assurance functions. With quantifiable quality measures at hand, one can evaluate software component, against predetermined critical values of the metrics.

Software metrics are the quantitative measures derivable from any attribute of the software life cycle in an algorithmic fashion [10]. Process of obtaining software metrics, **Software Measurement**, can be defined as a mapping from a set of objects in the software engineering world to a set of objects in the mathematical world [23]. Objects in the software enginnering world may be projects, products, and process. Objects in the mathematical world may be numbers or vectors of numbers.

Software metrics can be classified into three categories based on the measurable entities of software engineering life cycle [18].

Product metrics: metrics collected from explicit results of software engineering activities. e.g size, complexity of software product

Process metrics: metrics computed from the activities related to production of software engineering products. for example, effort, time, cost etc.

Resource metrics: metrics related to inputs to the software engineering activity. These inputs can be hardware, software, knowledge, and human resources.

1.1 Objectives and Scope

In this study only product metrics are investigated. All the metrics studied here, assume the conventional software development life cycle, and the third generation languages. Most of the metrics (except specification metrics) discussed here, measure

'complexity' of the respective software product, where complexity is psychological perception of difficulty for human being, in reading or writing the software. Formal notion of computational complexity, which measure arithmetic, or logical computations of algorithm is not of concern in this study. The larger the value of the metric, the more complex a program is supposed to be. It is assumed that, this *complexity* is related to various product qualities like *maintainability, testability, reliability*, etc. Product metrics, which measure other than 'complexity' attribute of the software product are not examined in this study.

The objective of this work is to make an extensive survey and analysis of all software metrics proposed in literature so far and to develop a few tools which automatically collect selected metrics from the respective software documents. These selected metrics have strong theoretical back ground, and some validation in the literature. Finally, experiments are conducted on three pilot projects to test the effectiveness of the metrics in predicting error prone components of the software. Correlations among the different software metrics are also computed for these projects.

1.2 Organization of the Thesis

Survey of various software metrics, that can be collected at different phases of software development life cycle is presented in chapter 2. Chapter 3 contains brief man-page like descriptions of the various tools developed to extract different software metrics. Detailed description of experiments conducted on the pilot projects, and analysis of results are given in chapter 4. Hypotheses that hold, or does not hold on these projects are also summarised in it. In chapter 5 the conclusions and the areas of further work are outlined.

CHAPTER 2

Software Metrics

In this chapter, we give a brief overview of various metrics that have been proposed for various products of a software life cycle.

2.1 Specification Metrics

Software requirements phase is generally the starting activity of a software development project. The main goal of this phase is to specify what the software is supposed to do. The output of this phase is the *Software Requirement Specifications*(SRS) document, which specifies all interfaces, functional specifications, etc. Two software metrics namely, Function Point Analysis and Bang metric are proposed in the literature to measure SRS document.

2.1.1 Function Point Analysis

Function point analysis is the earliest specification metrics which is proposed by Allen Albretch [2]. It is now perhaps the most commonly used specification metrics.

In Function point analysis, specification for a commercial data processing systems is used to calculate the size of a project at requirements specification phase. This

in turn is used to estimate cost and schedule of the project. Productivity of the personnel working in the project can also be measured with the size estimate given by function points. Function point analysis is based on the idea that one can measure the size of a system, in terms of the functions it delivers. This is done by counting the number of inputs, outputs, enquiries, master files, and interfaces in the system.

Information required for function point analysis can be easily gathered during the requirement analysis phase. The counts are then weighted, based on the function value to the customer. After a long process of trial and error, the author suggested some weights. They are

- number of inputs*4
- number of outputs*5
- number of enquiries*4
- number of master files*10
- number of interfaces*7

However, these weights can be tuned to reflect the particular environment. After adjusting these weights, sum of these weighted counts gives the function points delivered by the system. And then, this metric can be used to estimate cost and productivity as follows.

From the history of company's record, the number of function points of each project and the corresponding cost should be calculated. With these values, cost against the number of functional points should be plotted and a curve can be fitted. Then, we can use this graph to estimate cost of the project which is to be developed. Similarly, a graph of the number of hours per function point against the number of function points can be plotted. The number of hours per function point gives productivity index. It shows how much effort has been spent in delivering one function point.

There are two main advantages of function points. First, function points can be calculated at an early stage of a project. Second, it does not take much time to carry out a function point calculation. The major drawback is that, instead of taking weighted average, or total of the various counts, the weights should differ depending on the software to be computerised. Some projects may be functionally strong, for example robotics application, some may be data strong projects, like general enquiry program, and some may be hybrid. Instead of treating them differently, Functional Point analysis considers all the projects equally.

2.1.2 Bang

The major drawback of function point analysis mentioned above is the basis of this metric called *Bang* [8]. In this metric, software projects are divided into three domains, functionally strong, data strong, and hybrid. Then separate procedure to calculate *Bang* is proposed for each of these domains.

Projects are divided into the above mentioned domains based on the data collected from different views of the system. The system to be automated should be presented in the following three models.

- *Functional Model:* This is a characterisation of 'what system does ?' Typically, outcome of this is a *Data Flow Diagrams.*
- *Retained Data Model:* In this, system is analysed from the point of view of data inside the system. Product of this analysis is *Entity Relationship Diagrams.*
- *State Transition Model:* This is a representation of the behavioral states of the system. State Transition Diagrams are produced in this analysis.

Once the system is presented in these models, count of undivided elements in the data flow diagrams(FP), and count of relation ships(RE) in the retained data model

are calculated. The ratio (RE/FP) is used to decide the project as either functionally strong, data strong, or hybrid.

In the calculation of Bang metric for functionally strong systems, functional primitives(undivided elements of DFD) are used. Similarly, count of data objects are used in the calculating the *Bang* of data strong projects. In the hybrid case, system functions are divided into two, one primarily concerned with implementing system functions, and the other concerned with implementing the data organisation aspects of the system. Bang metric is calculated for each of the two projects. For a particular environment, these two bangs can be combined to form composite value.

2.2 Design metrics

Software design is a plan for a solution - a method by which SRS can be implemented. Typically, a design document is developed to specify the design. In the design document, various modules that makes up the system, and interconnections between them are specified. Modules may be interconnected via parameters, global variables, or function calls.

Software design metrics extracted from a design document, measure large scale properties of a system. Developers can use design metrics to uncover favorable and unfavorable design trends and to identify *stress points* that may lead to difficulties during coding and maintenance. *stress points* are critical components in the software system, which may cause trouble in the later phases of software development.

Given *Software Requirement Specification*, the system can be designed in many different ways, even assuming that all designs are modular. Some of these designs may lead to more interface errors that may come up during system integration, some may be easier to understand, some may lead to efficient implementation and low cost. Given two or more designs for a given problem specification, design metrics provides rational basis to chose the design with better quality.

Three factors which influences the design most are call dependency, cohesion, and coupling [1]. Several metrics are proposed to compare designs on these criteria.

2.2.1 Network Metrics

Network metrics measure the complexity of call dependency in the system design. Common calls and common access to database increases the coupling between the modules [22]. More the design deviates from a tree structure, the greater will be the difficulty in understanding the design. Deviation of structure chart from a tree *Graph impurity* can be measured as,

$$Graph\ \ impurity = n - e + 1$$

Where n, e are nodes and edges respectively in the structure chart.

One more work on network metrics is described in [9].The design metric suggested in it can be used to compare complexities of two systems with tree structured designs. Complexity of a tree structure T_c, is defined as

$$T_c = H * E * L$$

Where H is height of the tree, E number of branches of the root node, and L is the total number of leaves.

Another network metric is described by Benyon-Tinker [3]. Assuming pure tree structure, it calculates the metric based on the depth and breadth of the calling hierarchy.

2.2.2 Stability Metrics

The principle behind this type of metrics is that a poor system is one, where a change to one module has a high probability of giving rise to changes in other modules. This in turn, has high probability of of giving rise to further changes in other modules.

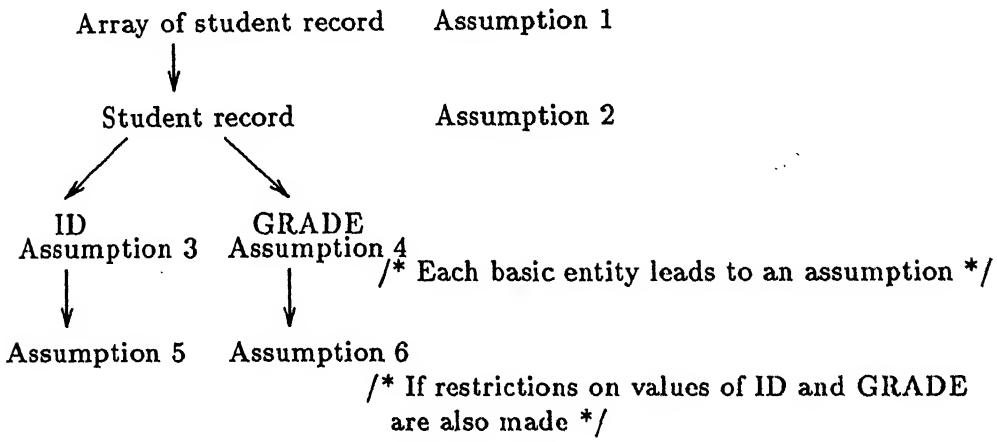

Figure 1: Assumptions made a module about 'array of student record'

Stability metrics measure the resistances to the potential ripple effect that occurs in the system, when a change is made in particular component.

One Stability metric, *Program Design Stability* is proposed in [21]. For each module in a system, the number of assumptions made by the other modules in the system, which either invoke the module, or share global data or files with the module, or invoked by the module, is calculated. This measure indicates the potential ripple effect that may occur, if the module is changed. *Program Design Stability* is calculated as the reciprocal of the total potential ripple effect of all its modules.

The term *assumptions made by a module about a parameter* can be illustrated with example as shown in fig. 1. If the parameter is 'array of student record', then the number of assumptions made by a module about the parameter is 6.

Another stability metric initially described in [16], is similar to the above metric. The effect of change on program unit is characterised by a probability matrix P, whose elements represent the probability that a change to a program unit will give rise to a change in another program unit directly.

P[i,j] = Probability that a change to program unit i will result

in a change to program unit j.

P[i,j] = 1, if (i=j)

As a change in one program unit having a probable effect on another program unit, there will also be second, third ... up to n'th order effect changes. The author demonstrated by simple algebra that the total effect of change can be characterised by the matrix,

$$(I - P)^{-1}$$

This can be used to estimate the resources required to implement changes to a software system. Major draw back of this work is, it is time consuming and difficult to estimate the elements of probability matrix.

2.2.3 Information Flow Metrics

These metrics measure the connections between modules in terms of the flow of information between them. A module, where there is a high information flow, requires more maintenance effort, or encourage errors to be made during implementation, operational, and maintenance phases. Information flow metrics can be used as software measures to analyse excessive coupling, between modules via parameters, global variables, and calls to modules.

Earliest work on information flow metrics was done by Henry and Kafura [13]. In this metric, it is assumed that the complexity of a module depends on two factors: intra module complexity and inter module complexity. Module size is used to represent intra module complexity. The inter module complexity is determined by the *inflow* and *outflow*, which represents the connections of a module to its environment. The inflow of a module is the number of abstract data entities flowing into the module. Similarly, outflow is the number of abstract data entities flowing out of the module. This leads to the following definition of module complexity.

$$Module\ Design\ Complexity = D_c = size * (inflow * outflow)^2$$

The term inflow*outflow represents the total possible number of combinations of input source to an output destination. This term is squared to reflect common experience that the difficulty in understanding, testing or modifying a system arises less from the action of a single component than from the interrelationships among the components.

Card and Glass [6] emphasised that *outflow* is more important than *inflow* because outflow indicates the functionality of a particular module. For that reason they defined the module complexity as,

$$Module\ Design\ Complexity = D_c = outflow^2$$

Further information flow metric, which is little variant to Henry and Kafura is proposed by Zage et. al [19]. In this, the design complexity of a procedure is defined as,

$$Module\ Design\ Complexity = D_c = call\_in * call\_out + inflow * outflow$$

Where *call_in* is the number of superior modules directly connected to the given module, and *call_out* is the number of subordinate modules directly connected to it.

The above discussed information flow metrics can be used to classify modules under consideration into either *normal*, *complex*, or *error prone*, based on the average and standard deviation values of the design complexities of modules in the system [19].

Assume *avg_cmplx* and *std_cmplx* as the average and standard deviation in design complexity values of modules in the system. Then, *cmplx_thresh* is,

$$cmplx\_thresh = avg\_cmplx + std\_cmplx$$

A module with design complexity C, can be decided as

- Error prone, if (C > cmplx_thresh)
- Complex, if (avg_cmplx < C $\leq$ cmplx_thresh)
- Normal, otherwise,

During the analysis done by Zage et. al [19], they found that the large scale software contain extreme outlier modules. Because of these extreme modules, threshold values are distorted. For this, *X-less design metric* algorithm is suggested. In this method, X modules are removed from the calculation of threshold values. By this new threshold values, more modules can be uncovered as potential trouble spots in the design. The number of modules to be ignored in the calculation of threshold values is up to the designer.

Once the stress points are identified, the designer can redesign the software and then recalculate the metrics to check the redesign. Furthermore, prediction of stress points at the design stage, will be helpful to the software managers in assigning difficult components to the most experienced developer, or to allow more time in testing the resulting code.

2.3 Source Code Metrics

2.3.1 Halstead's Software Science

The most straight forward and familiar measure for size of a Source code is *Lines Of Code* (LOC). This measures the physical size of the programs. The problem with using LOC, in using as a size measure is some lines are difficult to produce than other lines in the same program. To overcome this, Halstead [11] has suggested size measures based on the *operators and operands* in a program. A computer program is considered as a series of tokens, which can be classified as either *operators* or *operands*. These are considered as the basic elements of the given source program.

Operator is any symbol or keyword in program that specifies an algorithmic action. *Operand* is a symbol used to represent data. For example, arithmetic symbols, punctuation, semicolon, if then, do until, while do, GOTO, names of subroutines, and functions are operators in the program. Similarly, Variable names, literals,

labels are treated as operands.

The basic metrics in Hastead Software Science are defined as,

n_1 = number of unique operators.

n_2 = number of unique operands.

N_1 = Total occurrences of operators.

N_2 = Total occurrences of operands.

All the software science measures are functions of these basic metrics.

Vocabulary: This is a measure of number of unique elements that a programmer must deal with, to implement the program. Thus Vocabulary is defined as,

$$Vocabulary = n = n_1 + n_2$$

Length: Length is a measure of program size, and measures the number of times a programmer deal with each of the programming elements.

$$Length = N = N_1 + N_2$$

Estimated Length: Hasltead suggests that length can be estimated from vocabulary.

$$Estimated\ Length = N' = n_1 log(n_1) + n_2 log(n_2)$$

Some experiments showed that estimated length tends to be low for large programs and high for small programs [7].

Volume: Although length of a program can be considered as a measure of program size, Halstead suggests another approach that considers the number of times elements are used in a program and the total number of unique elements from which selections must be made thus,

$$Volume = V = N * log(n)$$

- LOC, Program Length and Volume are equally valid as relative measures of size [7].

Potential Volume: Among the different but equivalent programs, the one with minimal volume is defined as having the *Potential Volume*(V^*). According to Halstead, the minimal implementation of any algorithm can be done through a reference to a procedure that had been previously written. This requires invoking the procedures and the operands for it's inputs and output parameters.

$$Potential\ Volume = V^* = (2 + n_2^*)log(2 + n_2^*)$$

2 - represents the two unique operators for the procedure call, the procedure name and grouping symbol that separates the procedure name from it's parameters.

n_2^* - the number of conceptually unique input and output parameters.

This formula although useful for many programs, it is not applicable in all cases. Because, there are some programs that do not have explicit list of input/output parameters. For example, compiler, whose output consists of several files and messages to the operating system.

Difficulty: This is metric expresses the difficulty of reading or writing the code.

$$Difficulty = D = V/V^*.$$

Since V^* is hard to determine Halstead suggested alternate formula for difficulty.

$$D = \frac{(n_1 * N_2)}{(2 * n_2)}$$

This is a measure of *ease of reading or writing.* The reasoning behind the above formula is given as follows. The ratio $n_1/2$ indicates consideration of the difficulty in dealing with large number of operators. Since no program can be written with less than two operators (function call and end of statement). The ratio N_2/n_2 represents the average number of times operators are used. In a program where each operand is used only once, this ratio is 1. The more frequently a variable is changed in a program, the more difficult it is to retain it's current value in one's mind.

The reciprocal of *Difficulty* is defined as Programming Level.

$$Programming\ Level = L = 1/D = V^*/V$$

Programming Effort: It is the effort required to implement the program.

$$Programming\ Effort = E = V/L = D * V = \frac{n_1 * N_2 * N * logn}{2 * n_2}$$

Unit of measurement of E is elementary mental discriminations.

Programming time: Psychologists claim that human being is capable of making limited number of discriminations per second. This number (called Stroud number) ranges between 5 and 20. Thus, Programming time (T) of a program in seconds is defined as,

$$Programming\ Time = T = E/S$$

This formula can be used to estimate the programming time when given problem is solved by a single programmer writting a program.

Language Level: This metric measures the power of a language. Halstead hypothesised that if the programming language is kept fixed, then V increases, and L increases such that the product L * V remains constant. This product is defined as language level.

$$Language\ Level = \lambda = L * V = L * V$$

Some studies suggested that language level does not measure the language so much as it measures how the language is used in a program [7].

Information Content: This metric is a measure of the amount of function of a program. It is constant for a given algorithm regardless of language chosen to implement it.

$$Information\ Content = I = V/D.$$

2.3.2 Data Flow Metrics

Metrics under this classification measures the usage and visibility of data as well as their interactions.

Live Variables

This Metric is based on the hypothesis is that "more the data items that a programmer must keep track of, when constructing a statement, the more difficult it will be to construct it". Average number of live variables per statement ($\bar{LV}$) can be used to capture intra module data usage. A definition of *live variable* is "A variable is live from it's first reference to the last reference". Average live variables per statement is the sum of count of the live variables divided by the count of executable statements in a procedure.

Variable Span

Span size captures how often a variable is used in a program. *span* is the number of executable statements between successive references to the same variable. For a program that references the same variable in n statements, there are (n-1) spans for that variable. A large span require the programmer to remember a variable that was used far back in the program. Average span size is the average number of executable statements that pass between successive references of variable.

Segment Global Usage Pair

This metric measure the program complexity based on the usage of global data within a program. A segment global usage pair (P,r) is used to represent the instance of a module P, using the global variable r. The Actual Usage Pair(AUP) is the number of actual 'segment global usage pairs' in the program, and Potential Usage Pair(PUP) represents the number of times a module could access a global variable. Then relative usage pair RUP is defined as,

$$RUP = AUP/PUP$$

This measure gives a probability that a module will reference a global variable in the program.

Program Slicing

Program Slice is a reduction of original program, which represents it within the domain of the specified behavior [20]. And this specified behavior is called slicing criterion. Here, an example to show how slices of a program are made.

Original program.

```
1  begin
2       read(x,y);
3       total:=0.0;
4       sum:=0.0;
5       if (x ≤ 1) then
6            sum := 4;
7       else begin
8                 read(z);
9                 total:=x*y;
10           end
11      write(total, sum)
12 end.
```

Suppose slice on the value of z at statement 12 is to be made. Value of z is modified in statement 8, execution of this statement depends on the condition in statement 5. Statement 5 checks the Value of x, and this value is modified in statement 2. Thus the slice on the value of z at statement 12 is,

```
1  begin
2       read(x,y);
5       if (x ≤ 1)
```

```
6       then
8       else read(z);
12  end.
```

Similarly, slice on the value of total at statement 12 is,

```
1   begin
2       read(x,y);
3       total:=0;
5       if (x ≤ 1)
6       then
9       else total :=x*y;
12  end.
```

Similar to the above, we can have slices on each variable. The following are the few slicing based metrics, proposed in the literature.

Coverage: Ratio of mean slice length to program length. Low coverage value indicates large program with many short slices and implies that the program has different conceptual purposes.

Overlap: This is computed as the mean of the ratio's of non-unique to unique statements in each slice. A high overlap indicates very interdependent code.

Parallelism: It is the number of slices which have few statements in common. A high degree of parallelism suggests that assigning of a processor to execute each slice in parallel can give a significant speedup.

Tightness: This is a ratio between the number of statements which are in every slice to the total program length. High tightness indicates that all the slices in a subroutines are really belonged together. This can be used as a measure of cohesion.

2.3.3 Control Flow Metrics

These metrics measures the comprehensibility of program flow charts.

Cyclomatic Complexity

Cyclomatic complexity number, v(G) is a measure of linearly independent paths in a program flow chart [15]. For a program flow chart with e edges, and n nodes,

$$v(G) = e - n + 2$$

Cyclomatic complexity number can also defined as, the number of edges, that should be removed to reduce the flow graph to it's skeleton - that is one without "circuits" and "loops". For the programs made of structured constructs, having single entry and single exit points, V(G) is one plus the number of decisions.

Myer's extension to Cyclomatic Complexity

Myer [14] has extended the notion of cyclomatic complexity considering the decisions with multiple conditions. In this extension, complexity is measured as an interval rather than a single value. The lower bound of the interval is the number of decisions plus one, and the upper bound of the interval is the number of individual conditions plus one.

Gilb's Logical Complexity

Gilb [14] proposed two control flow metrics.

1. Absolute logical complexity, C_L - Number of binary decisions.

2. Relative logical complexity, c_L - Ratio of C_L to total number of statements .

This is an improvement over pure control metric as it takes into account some volume metric.

Knot Count

Knot Count is a measure of control jumps overlap in a program. This can be used to measure the uncontrolled usage of GOTO statements in a FORTRAN program.

Average Nesting Level

In this metric, average nesting level of the executable statements in a program is calculated. Higher the average nesting level, more difficult it is determine entrance conditions for the statements in the program.

Minimum number of paths (N) and Reachability(R)

- Minimum number of paths (N) - Number of distinct execution sequences from entry node to exit node of a flow chart.
- Reachability of a node (R) - Number of ways of reaching a node of the flow chart from it's entry node.

Average reachability , is an average of reachability values for each node in a flow chart.

2.3.4 Hybrid Metrics

Several metrics are proposed by researchers to measure combinedly, both data flow and control flow aspects of the program.

Hansen's : Hansen's [12] metric combined the cyclomatic complexity metric and one of the Halstead's metric. It consists of a pair (cyclo,op) where op is count of operators in the program.

Oviedo's : Oveido [17] proposed a composite program complexity metric (C), based on the control flow complexity and a data flow complexity. In it's simplest form,

$$C = \alpha * CF + \beta * DF$$

for some values of α and β.

2.3.5 Programming Style

While implementing the software design of a system in any programming language, the programmer has to follow certain guide lines to make the source code easy to read and understand. Some attempts have been made to measure, how far a programmer is following the specified programming guidelines. In this section, we discuss a systematic procedure to measure the programming style of a programmer [4].

Each program is given a percentage of *score* for style that consists of contributions in varying degrees from the following program features:

Module length: The average length, in non blanks, of module definitions. A low figure implies a well structured, understandable program.

Identifier length: The average length, of user identifiers; This gives an approximate indication of the use of meaningful names. Readability increases with longer identifier lengths.

Comments: The percentage of all lines containing comments; Understandability of the code increases with more comments. But after certain point, excessive usage will obscure the program statements.

Indentation: The ratio of initial spaces to total number of characters ; Here also, a high figure is is taken to mean a better style.

Blank lines: The percentage of all lines that are blank; Blank lines are one of the easiest ways of providing visual clues to the program structure. This measure provides an indication of how well the program structure is made visible to the reader. A high figure indicates a more readable program.

Line length: The average number of nonblank characters per line; It is assumed that the higher the value of this measure, the less readable the program. There must also be a lower bound, as a very low figure also implies a program which is difficult

to read.

Embedded spaces: The average number of embedded spaces per line; These embedded spaces highlight the intra statement structure. This is useful particularly within arithmetic and logical expressions, data type declarations. The higher the value for this measure, the better the style.

Constants definitions: The percentage of all user identifiers that are defined as constants.

Reserved words: The number of different reserved words and standard functions used; The assumption is that the higher the this value, the better the use of the range of facilities available to the programmer.

Include files: The extent to which a program is segmented by using #include files;

goto's: The number of occurrences of a goto statement; A zero figure is assumed to be the best style.

A score is associated with each metric defined above. Each contributes to a different maximum percentage to the final score, and each is additive ,with the exception of the last which is subtractive.

For each metric, five parameters are provided which define a conversion curve as shown in fig 2.3.5.

The parameter 'max' specifies the maximum percentage score to be given for each measure. If the value of the measure lies in the range defined by lotol and hitol, then maximum score for that measure is given. Similarly, if measured value fall in either of the ranges(low,lotal) or (hitol,high) then a linear proportion of the maximum score is assigned. When the value of the measure is less than low or greater than high, a score of zero is obtained. The scores for each measure are summed except that of last (goto's), which is subtracted from the total. The conversion graph shows that too high or too low figure for each metric decreases the final score, since these values indicate poor style.

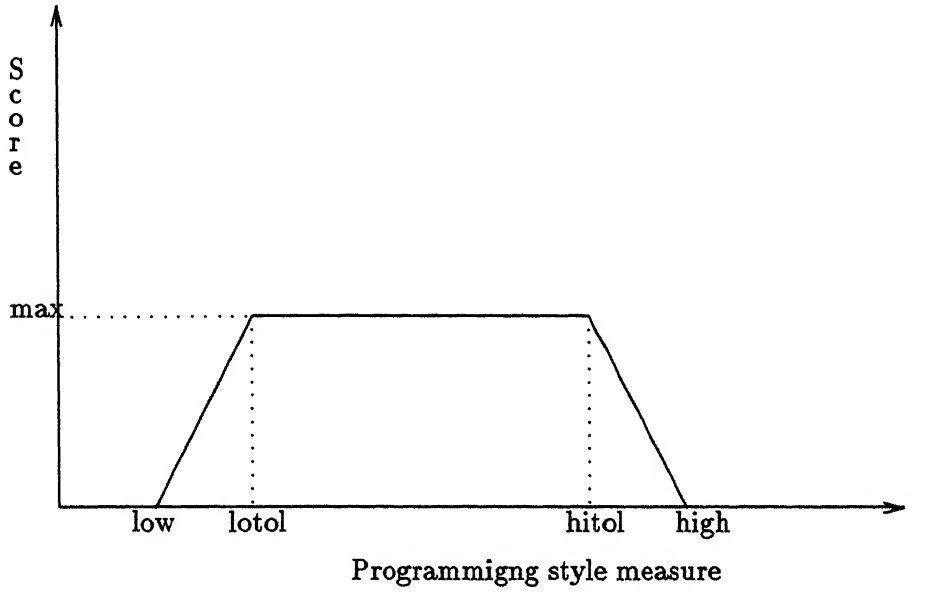

Figure 2: Metric Value Calculation

The fig 3 provides values of parameters to each metric.

A score above 60 percent can be graded as properly presented program, and a score below 20 percent as very poor.

Metric	max	low	lotol	hitol	high
Module length	15	4	10	25	35
identifier length	14	4	5	10	14
Comment lines(%)	12	8	15	25	35
Indentation(%)	12	8	24	48	60
Blank lines (%)	11	8	15	30	35
Characters per line	9	8	12	25	30
Spaces per line	8	1	4	10	12
Defines (%)	8	10	15	25	30
Reserved words	6	4	16	30	36
Include files	5	0	3	3	4
Gotos	-20	1	3	99	99

Figure 3: Metrics Boundary Values.

CHAPTER 3

TOOLS

A number of tools were developed to collect various metrics. In this section, a brief man-page like description of the various tools is given.

3.1 Tool for design complexity from design specs.

NAME

dmetric - computes metrics for program designs to spot error prone or complex modules or subsystems.

SYNOPSIS

dmetric [-d] [-s] [-x] filename

DESCRIPTION

dmetric calculates design complexity of program designs. The design complexity of each module is computed as (in_flow*out_flow + call_in*call_out), [ref 1.] where

- in_flow - no. of abstract data entities flowing into the module.
- out_flow - no. of abstract data entities flowing out of the module.
- call_in - no. of modules calling this module.
- call_out - no. of modules called from this module.

Note that, in call_in & call_out, calls to the library functions are not included. Further more, a function is assumed as a library function if it's definition does not exists in the input program design file.

Based on the design metric, dmetric classifies modules as *Error prone* or *Complex*. *Error prone* modules are those, whose complexity is more than one Standard deviation above the mean. *Complex* modules are those whose complexity is more than the mean, but less than mean plus standard deviation. Alternatively, for defining *error prone*, and *complex* some fixed threshold values can be used. For this approach, first change the CMPLX_THRESHOLD & ERR_PRO_THRESHOLD values and set FIX_MOD in the file constants.h., then recompile using makefile.

To remove the distortions produced by the extreme outlier modules, in average standard deviation values, *x-less* algorithm can be applied. In this algorithm, X modules are ignored while calculating average and standard deviation values. By this new threshold values, more modules can be uncovered as potential trouble spots.

To use Other design complexity metrics such as Kafura metric[ref 2], Card metric[ref2], definition of METRIC_FORMULA in 'constants.h' should be changed and recompiled using makefile.

OPTIONS

-d detailed output. It produces the call_in, call_out, in_flow & out_flow for each module.

-s sorted output. Design complexities of the modules will be listed in the decreasing order.

-x *x-less* algorithm is applied in calculating the threshold values.

DESIGN SPECIFICATION LANGUAGE

The design specification is a simple language, that has been designed keeping C programming language in mind. The grammar for design specification language in BNF notation is given in file syntax.

The design specification consists of two parts. Data definition and module definition. Data definition contains the definitions of structures that are used later in declarations of module parameters and is specified as in C programming language.

example: struct int_node

```
{
   int index;
   struct int_node *next;
};
```

Module definition follow the data definition. For each module, it's name, parameters, subordinates, and size should be defined in the specified format.

example:

```
int get_validated_data(IN:file1, file2)
/* IN indicates parameter is input to the module,
* similarly OUT, IN_OUT should be used */
char *file1, *file2;
{
   SUBORDINATES:validate_file1(), validate_file2();
   SIZE: 50
}
```

EXAMPLES

To find design metrics from 'input' file

```
example% dmetric input
```

If detailed output is required

```
example% dmetric -d input
```

Now look 'input.metrics' file for the results.

WARNINGS

dmetric does not do extensive error checking, program will be terminated at the first syntax error. Only, warning messages will be printed for the modules which are called, but definition for which do not exist in the modules specifications file.

REFERENCES

1. "Evaluating design metrics on large-scale software". Wayne M.zage, Dolores M.zage. IEEE Software July, 1993.

2. "The Evaluation of software system's structure using quantitative software metrics". Henry.S, Kafura.D. Software Practice & Experience June 1984.

3. "An evaluation of software design using the DEMETER tool" Al-Janabi, Aspinwall.E. Software Engineering Journal Nov, 1993.

3.2 Tool for design complexity from source code

NAME

dmetre - computes metrics for design of C programs to spot error prone or complex modules or subsystems.

SYNOPSIS

dmetre [-d] [-s] [-x] [-m] filename1 filename2 .....

DESCRIPTION

dmetre calculates design complexity of C programs. The design complexity of each module is computed as (in_flow*out_flow + call_in*call_out),[ref 1.] where

in_flow	-	no. of abstract data entities flowing into the module.
out_flow	-	no. of abstract data entities flowing out of the module.
call_in	-	no. of modules calling this module.
call_out	-	no. of modules called from this module.

Note that, in call_in & call_out, calls to the library functions are not included. Further more, a function is assumed as a library function if it's definition does not exists in the input C program file.

Based on the design metric, **dmetre** identifies modules that are *Error prone* or *Complex*. *Error prone* modules are those, whose complexity is more than one Standard deviation above the mean. *Complex* modules are those whose complexity is more than the mean, but less than mean plus standard deviation. Alternatively, for defining *error prone*, and *complex* some fixed threshold values can be used. For this approach, first change the CMPLX_THRESHOLD & ERR_PRO_THRESHOLD values and set FIX_MOD in the file constants.h., then recompile using makefile.

To remove the distortions produced by the extreme outlier modules, in average standard deviation values, *x-less* algorithm can be applied. In this algorithm, X modules are ignored while calculating average and standard deviation values. By

this new threshold values, more modules can be uncovered as potential trouble spots.

To use Other design complexity metrics such as Kafura metric[ref 2], Card metric[ref2], definition of METRIC_FORMULA in 'constants.h' should be changed and recompiled using makefile.

OPTIONS

-d detailed output. It produces the call_in, call_out, in_flow & out_flow for each module.

-s sorted output. Design complexities of the modules will be listed in the decreasing order.

-x *x-less* algorithm is apllied in calculating the threshold values.

-m input files in a file. If many input files are there, then all the input files can be listed in a file and that file can be given as input file with this option.

EXAMPLES

To find design metrics from code.c & main.c

example% dmetre code.c main.c

If detailed output is required

example% dmetre -d code.c main.c

Suppose 'inputfiles' contains list of input files as

main.c code.c y.tab.c

ld2/mantri/proj/struct/lex.yy.c

then to extract design metrics from all these files,

example% dmetre -m inputfiles

Now look 'metrics' file for the results.

WARNINGS

dmetre does not do extensive error checking, so properly compiled input C program files are expected. The program will be aborted at the first syntax error.

If the modules of the system, for which design metrics are to be calculated are spread over multiple files, then all those files should be given as input. Otherwise, some modules for which definitions does not exists, are considered as library functions. But if a file is already included in another file, then there is no need to give it as input file again. This tool extracts metrics only after all the preprocessor statements are expanded using C preprocessor("/usr/lib/cpp").

REFERENCES

1. "Evaluating design metrics on large-scale software". Wayne M.zage, Dolores M.zage. IEEE Software July 1993.

2. "The Evaluation of software system's structure using quantitative software metrics". Henry.S, Kafura.D. Software Practice & Experience June 1984.

3. "An evaluation of software design using the DEMETER tool" Al-Janabi, Aspinwall.E. Software Engineering Journal Nov,1993.

3.3 Tool for source code metrics

NAME

halstead - computes halstead software science measures and cyclomatic complexity for C programs.

SYNOPSIS

halstead [-c] [-h] [-s] [-d] [-m] filename1 filename2 ...

DESCRIPTION

halstead calculates halstead's metrics and cyclomatic complexity values of each module. Cyclomatic complexity is the number of linearly independent paths in the control flow graph of a module. For structured programming languages having single entry and single exit constructs, Cyclomatic Complexity is one plus the number of decisions.

Halsteads's metrics are based on four basic measures as, number of unique operators n_1, number of unique operands n_2, the total number of operators N_1, the total number of operands N_2. *Operator* is any symbol or key word in a program that specifies an algorithmic action. Similarly, *Operand* is a symbol used to represent data.

All the software science measures are functions of these basic metrics and they are defined as follows.

Volume(V)	=	$(N_1 + N_2)$*log$(n_1 + n_2)$
Difficulty(D)	=	$(n_1$*$N_2)/(2$*$n_2)$
Programming Effort	=	(V * D)
Information Content	=	V/D

Volume metric can be used as a measure for program size. Difficulty metric expresses the difficulty of reading or writing the code. Programming Effort captures the effort required to implement the program. Information Content measure the

amount of function of a program. It is constant for a given algorithm regardless of language chosen to implement it.

OPTIONS

-c Outputs cyclomatic complexity, and size of each module.

-h Outputs halstead measures Volume, Difficulty , Programming Effort, and size of each module.

-s Produces the output in decreasing value of corresponding metrics.

-d detailed output. It produces all the Software science measures for each module.

-m input files in a file. If many input files are there, then all input files can be listed in a file and that file can be given as input file with this option.

EXAMPLES

To find halstead measures from code.c & main.c

example% halstead -h code.c main.c

To find Cyclomatic complexity values from code.c & main.c

example% halstead -c code.c main.c

If detailed output of Halstead measures is required

example% halsted -d -h code.c main.c

Suppose, 'inputfiles' contains list of input files as

main.c code.c y.tab.c

ld2/mantri/proj/struct/lex.yy.c

then to extract halstead metrics from all these files,

example% halstead -h -m inputfiles

Now look 'halstead_metrics' file for the results.

WARNINGS

halstead expects properly compiled C programs. The metrics produced by this

tool may be dubious, if syntactically incorrect program is given as input.

REFERENCES

1. "A complexity measure".McCabe.T.J. IEEE TOSE, Dec 1976 .

2. Halstead.M. Elements of Software Science.1977.

3. "A perspective on software science".Christensen K.G, Fitsos.P, Smith.C.P. IBM System's Journal.Oct'81.

3.4 Tool for live variables and span size analysis

NAME

live - computes average live variables per statement, and average span size of variables for C programs.

SYNOPSIS

live [-m] filename1 filename2 ....

DESCRIPTION

For each subroutine of C programs, live calculates average live variables per statement, and average span size of variables.

"A variable is *live* from it's first to last reference within a procedure". The average number of live variables per statement is defined as "the sum of count of live variables divided by the count of executable statements in a procedure". In this tool, live variables are computed based on the order of statements in the source program, rather than run time order.

Span is the number of statements between two successive references to the same variable. For a procedure that references a variable in n statements, there are (n-1) spans for that variable. Average of these spans is Average span size of that particular variable. live tool outputs the average of "average span sizes of variables" referenced in a particular procedure.

OPTIONS

-m input files in a file. If many input files are there, then all input files can be listed in a file and that file can be given as input file with this option.

EXAMPLES

To find live variables and span size from code.c & main.c

example% live code.c main.c

Suppose, 'inputfiles' contains list of input files as

main.c code.c y.tab.c

ld2/mantri/proj/struct/lex.yy.c

then to extract metrics from all these files,

example% live -m inputfiles

WARNINGS

live expects properly compiled C programs. Metrics produced by this tool may be dubious, if syntactically incorrect program is given as input.

REFERENCES

1.Conte, Dunsmore, Shen. "Software enginering metrics and models".

3.5 Tool for Programming Style

NAME

style - computes style measures for C programs and gives grading for programmers style.

SYNOPSIS

style -[d] filename

DESCRIPTION

style evaluates how far the programmar is following the programming guide lines. Program is given a percentage of "score" for style that consists of contributions in varying degrees from different programming guide lines that are to be followed by the programmer.

Programming style features that are considered by this tool are identifier length, comments, indentation, blank lines, average line length, embedded spaces, constants definition, usage of include files & goto's, and module length. A score is associated with each feature mentioned above. Each contributes to a different maximum percentage to the final score, and each is additive , with the exception of usage of goto's which is subtractive. Too high or too low usage of above features decreases the final score since that indicates poor style.

A score above 60 percent represents well written program, where as a score below 20 percent indicates very poorly presented program.

OPTIONS

-d detailed output. It produces style measures for each module.

EXAMPLES

To find style measures from code.c

```
example% style code.c
```

If detailed output is required,

example% style -d code.c

WARNING

style expects syntactically correct C programs. The results produced by this tool may be dubious, if unproperly compiled C program is given as input.

REFERENCES

1. "Automatic assessment aid for Pascal programs". Michael J.Rees. ACM SIGPLAN, 1981.

2. "A Style Analysis of C Programs". Berry.R.E., Meekings.B.A.E. CACM Jan,1985.

CHAPTER 4

Experiments

In this chapter, we describe the experiments that we performed on various projects using the tools developed.

4.1 Experiment Objectives

The basic goal of the experiments is to study if metrics can be used to improve the quality of the final software. For quality, we focus mostly on errors, as these effect the primary objectives of a software the most. The primary objective of Quality Assurance is to ensure that software has minimum number of errors left at the time it is delivered, (i.e. maximum number of errors are caught before delivery), and that errors are detected with the least cost.

For both the reasons, we would like the error prone areas of software to be identified as soon as possible, but definitely before testing. Such information, if available early, can be effectively used to reduce cost for improving the quality. For example, if error prone modules can be identified at the design stage, this information can be used to assign the error prone modules to more experienced programmers. If error prone modules are identified after coding (but before testing), then this information can be used for improving quality, by requiring more stringent Quality

Assurance criteria for those modules e.g. necessary reviews of these modules by experienced people, 100% branch coverage of these modules, necessary reviews of the test plan, and test cases, etc.

Hence, the basic objective of the experiments is to test existing hypothesis or develop new ones for identifying error prone modules at design time, or after coding is done, but before testing is performed.

Another related objective is to see if the complexity of modules can be effectively reduced by using metrics to highlight complex or problematic modules. High complexity is a major contributor to high maintenance cost. It follows that if the design is kept simple and modular i.e. the complexity of the design is less, then maintenance will be easier. We work with a philosophy that design is a creative process that cannot be automated. However, it is important to be able to effectively, and objectively evaluate the quality of a design. This evaluation can be used as a feed back by the designer to improve the design. As mentioned above, the the quality measure we take is complexity - the simpler the design the better, it is. If a basis for evaluation can be developed and validated, then a tool implementing this can be invaluable during design. The designer can design and then check the quality of his design. As of now, mostly quality is checked through reviews, which are cumbersome, and time consuming.

4.2 Experiment Methodology

For our experiments, we have used three different projects developed in C belonging to three different application areas. The following is a brief descriptions of the projects considered for experiments.

- Project A: *Design and flat pattern development of transition surfaces.* In this project, development of various types of surfaces like, transition sections,

Characteristic	Project A	Project B	Project C
Application area	computer graphics	information retrieval	timetable scheduling
Total size (LOC)	3500	2108	1332
Number of modules	172	68	36
Average module size	20	31	37

Figure 4: Characteristics of Projects

canonical convolutes, and branch fittings are automated. Implementation of this work is carried out on HP Turbo Srx Workstation using Starbase Graphics utilities.

- Project B: *Information retrieval in Indian Languages.* Some tools necessary for linguistic support are implemented in this work. These tools are then used to extend existing distributed information retrieval system to support natural language text retrieval in a regional language. Implementation of this project is done on SUN workstations.

- Project C: *Timetable scheduling.* Given a list of "course offered, it's expected enrollment, and a set of time preferences given by the instructor", it schedules the courses in the rooms within some specified constraints. Out of the three projects considered for experiments, this is the only project, where water fall life cycle is strictly followed. SRS and Design documents were produced and available. This is one of the reasons to consider this project in experiments, even though size and complexity of this system is less, when compared to the other two projects. This project too is implemented on SUN workstation.

For each Project, the complete C source code was available. From the source code, using our tools, we are able to collect various metrics data like design metrics, cyclomatic complexity, Halstead metrics, live variable analysis ... etc.

For the Project C, the design document, and the SRS were also available. The language used for design description was consistent with our tool for extracting design metrics from the design document. Besides this project, for all the other projects only source code was available, which was used for extracting the design metrics also (we have a tool for this).

We also obtained data about the errors of the different projects. For each project, the errors found in the different modules were compiled. The errors were classified by the programmer as *Hard*, *Medium*, and *Easy*. *Hard* errors are those which took more than 8 hrs. to locate and fix. Similarly, *Medium* difficult errors are classified as errors which took in between 1 to 8 hrs to locate and fix. Errors, which took less than 1 hr., are classified as *Easy*. These *Hard, Medium, and Easy* errors were given the weights 8, 4, and 1 respectively while calculating the total errors in a module or project.

Apart from error data, subjective opinion of the programmer about the modules that they had developed is also collected. The programmer is asked to classify the modules in the system strictly based on their psychological perception of difficulty in writing the module. It is important to note that this subjective opinion can not be used to compare modules across different projects, because these opinions are given by different people. However, this can be used for comparison with in a particular project. Based on the subjective opinion, modules are classified as *Hard, Moderate, and Easy*. There are no clear cut definitions, in defining these, but classification is done so that complexities of modules in a particular category, are comparable. If a module has been classified as either Hard, or Moderate through subjective opinion, we refer it as *module highlighted by the programmer*.

Characteristic	Project A	Project B	Project C
Total number of modules	172	68	36
Modules having above mean errors	27	19	9
Total errors found	889	482	66

Figure 5: Error data of Projects

4.3 Data Analysis

4.3.1 Programmer Perceived Complexity and Errors

Performance of subjective opinion is shown in figure 6. We found that subjective opinion of complexity of various modules, as given by the programmer was very good at predicting error prone modules. As mentioned above, we consider all the modules ranked as Hard and Moderate, as having been *highlighted.* As can be seen from the data, in all the three different projects, the modules that are highlighted contained around 80% of total errors found in that particular project. Furthermore, as can be seen from the data in all the three projects, very few modules are highlighted that do not have above average errors, and very few modules having above average errors have not been highlighted.

Hence, it is clear from this data that programmers subjective perception of complexity correlate very well with errors, and is a very effective way to accurately identify error prone modules(i.e. the one with above average errors).

One of the reasons for this could be that software complexity arises from different factors. Since opinion is given by the programmer, we can assume that it captures all the different factors from which complexity arises. On the other hand different metrics proposed in the literature, capture different aspects of the software. Furthermore, as complexity is finally a psychological or perception concept, which the different metrics try to qualify, the subjective evaluation is indeed closest to the

actual concept of complexity. This type of subjective opinion also takes into account the capability of a programmer. One programmer may find a module, which another may find moderate or easy. From the point of view of error proneness, it is important that this difference be captured, since the programmer that finds the module hard is likely to make more errors, hence the module should be highlighted. This type of discrimination cannot be done by any metrics, they can not capture programmers capability.

So it is clear that a simple classification by the programmer is a very effective way to highlight error prone modules. The main disadvantage of subjective opinion is that they may become biased to obtain a desired result. In our experiments, there was no reason for such bias as the evaluation of data had no bearing on the project or programmer. But, in production environments where this data may be utilised for evaluation, biases may creep in. So for such a measure to be useful, it is essential to obtain unbiased feed back.

Another limitation of this method is, that it does not seem that subjective opinion can be effectively given at design time. Though, we had only one project in which design phase was formally separated from coding, the feed back from the designer(who was also the programmer) was that it is hard to judge complexity of modules at design time. This limitation is there with many code based metrics also. However, it would seem possible that with experience, programmer may be able to give complexity evaluation at design time which will be very close to their evaluation after coding. However, further experiments need to be done to validate this.

4.3.2 Design metrics and Relationship to Errors

Three different information flow metrics were proposed in the literature so far. These metrics can be used to classify modules under consideration into *error prone, complex*, or *normal.* We use three different methods for classifying the modules. These

Characteristic	Project A	Project B	Project C
1. Modules highlighted by the programmer.	24	19	11
2. Modules highlighted, but having less than mean errors	1	1	2
3. Modules having above mean errors, but not highlighted.	4	1	0
4. % of modules highlighted.	13	27	30
5. % of total errors in the highlighted modules.	80	80	81

Figure 6: Performance of Subjective Opinion in Predicting Errors

are described below. The objective here is to find out, which of them has better performance in predicting errors. Here after, we refer *complex* and *error prone* modules as *modules highlighted by the design metrics.*

Assume *avg_size* and *avg_cmplx* are the average values of module sizes and module design complexities respectively. Similarly *std_size* and *std_cmpx* are standard deviations in module sizes and module design complexities respectively. *size_thresh* and *cmplx_thresh* are defined as,

$$size\_thresh = avg\_size + std\_size$$

$$cmplx\_thresh = avg\_cmplx + std\_cmplx$$

Method A: [19] A module with design complexity C, is decided as

- Error prone, if (C > cmplx_thresh)
- Complex, if (avg_cmplx < C $\leq$ cmplx_thresh)
- Normal, Otherwise

Method B : In some cases, method A predicts a module as error prone, even though it is very simple. This occurs in case of coordinate modules. The primary concern

of these modules is the flow of data to and from different subordinates. Estimated size of a module at design stage can be used to avoid this kind of false positives. Further, correlation studies between size and number of the errors found suggest that the size of the module should also be considered in predicting a module as error prone. Module size gives rough estimation of intra modular complexity, where as design complexity measures only inter module complexity. Hence, the module size is incorporated as follows. A module with size **S**, and design complexity **C**, is

- Error prone, if (C > cmplx_thresh) **and** (S > size_thresh)
- Complex, if (avg_cmplx < C $\leq$ cmplx_thresh) **and** (S > avg_size) **or** (S > size_thresh)
- Normal, Otherwise

Method C: [19] In this method, modules are highlighted as in method B, except that *x-less* algorithm is also applied. In this, extreme module size values, and design complexity values are ignored in the calculation corresponding average and standard deviation values. Feed back from the user is needed to determine which module should be ignored.

We used **dmetre** tool to extract design information from source code of all the three software projects. Normally, the designer would calculate the design metrics as the design of the system completes. Since, we were experimenting with completed projects, we had to extract design information from the source code. However, another tool we developed **dmetric**, could be used to compute design metrics from the program design specifications.

Performance of various design metrics in predicting the error proneness of software are shown in the tables 7, 8, 9. Characteristics of the results shown in tables 7 and 8 are similar, and can be summarised as follows.

- modules highlighted, but having less than average errors, are decreasing from method A to method B in all design metric formulas. This strengthens our

Data characteristic	*Zage metric*			*Card metric*			*Kafura metric*		
	Methods			*Methods*			*Methods*		
	A	B	C	A	B	C	A	B	C
1. Modules highlighted by dmetre.	18	20	28	12	14	26	10	13	23
2. Modules highlighted, but having less than mean errors	3	2	6	3	1	7	1	1	4
3. Modules having above mean errors, but not highlighted.	12	9	5	18	14	8	18	15	8
4. % of modules highlighted.	10	11	16	6	8	15	5	7	13
5. % of total errors in the highlighted modules.	57	70	79	40	58	72	43	56	72

Figure 7: Performance of dmetre on Project A

Data characteristic	*Zage metric*			*Card metric*			*Kafura metric*		
	Methods			*Methods*			*Methods*		
	A	B	C	A	B	C	A	B	C
1. Modules highlighted by dmetre.	21	17	21	23	19	21	7	11	14
2. Modules highlighted, but having less than mean errors	10	4	6	12	4	6	2	1	2
3. Modules having above mean errors, but not highlighted.	8	6	4	8	4	4	14	9	7
4. % of modules highlighted.	30	25	30	33	27	30	10	16	20
5. % of total errors in the highlighted modules.	56	64	75	62	73	74	28	52	62

Figure 8: Performance of dmetre on Project B

claim that rough estimation of module size at design stage can be used to eliminate modules, which are having high design complexity values, but not error prone.

- *x-less* algorithm (Method C) is able to uncover more error prone modules. Even though, this algorithm is increasing the number of false positives slightly, it is less significant when compared to the number of extra uncovered error prone modules.

- Modules having above average errors, but not highlighted are decreasing from Method A to Method C in all design metric formulas. And this number is least in Method C, using *Zage design metric* formula.

- Considering the total number of errors in highlighted modules, performance of dmetre is increasing from Method A to Method C, and Kafura metric to Zage metric. Application of Method C, with Zage metric dmetre, was able to pinpoint 79% of errors, and highlighted only 16% of modules in Project A. Similarly, in Project B, 30% of modules were highlighted, and these contain 75% of errors.

- Performance of dmetre in highlighting the errors, increased significantly (by 13% in Project A, and by 24% in Project B), from Method A to Method C, with Kafura design metric formula. In Method A, deciding error proneness of the module is done only based on design complexity values, where as in Method B, module sizes are also considered. But, Kafura metric already includes module size in it's formula. So, we claim that, the way in which module size is added in Kafura metric formula is not suited.

Inferences from the results obtained on Project C.

Data characteristic	*Zage metric*			*Card metric*			*Kafura metric*		
	Methods			*Methods*			*Methods*		
	A	B	C	A	B	C	A	B	C
1. Modules highlighted by dmetre.	10	10	10	10	9	10	6	8	8
2. Modules highlighted, but having less than mean errors	3	2	3	3	2	3	0	1	3
3. Modules having above mean errors, but not highlighted.	2	1	2	2	2	2	3	2	4
4. % of modules highlighted.	27	27	27	27	25	27	16	22	22
5. % of total errors in the highlighted modules.	69	69	72	69	69	72	54	68	68

Figure 9: Performance of dmetre on Project C

- There is no significant performance improvement, while using *x-less* algorithm(Method C) in this project. This is because, it does not have any significant extreme outlier modules.

- Similar to the results of table 7, 8, in this case also, performance of dmetre in predicting errors increased significantly (by 14%) from Method A to Method C, using Kafura design metric.

- On the overall, performance of design metrics is same in all methods. This is due to the relatively small size of the project, and inherently less complex nature of the problem.

Since, the design document is available for project C, we also computed design metrics from it using dmetric tool. Figure 10 gives various statistics related to the performance of dmetric on this project. It highlighted less number of error prone modules when compared to the modules highlighted by the dmetre, which computes design metrics from the source code. This is because some minor data structures are ignored, while specifying the design. As a result of this, inflow and outflow values

Data characteristic	*Zage metric*			*Card metric*			*Kafura metric*		
	Methods			*Methods*			*Methods*		
	A	B	C	A	B	C	A	B	C
1. Modules highlighted by dmetric.	8	6	6	11	6	6	6	6	6
2. Modules highlighted, but having less than mean errors	5	0	0	5	0	0	3	0	0
3. Modules having above mean errors, but not highlighted.	6	3	3	3	3	3	6	3	3
4. % of modules highlighted.	22	16	16	30	16	16	16	16	16
5. % of total errors in the highlighted modules.	43	60	60	63	60	60	43	60	60

Figure 10: Performance of dmetric on Project C

of a module are decreased, which in turn reduced the the design complexity values of the module. Otherwise, the performance of dmetric could have been same as that of dmetre.

Correlation of Errors with different Design measures

Correlation coefficients between different metrics are obtained to derive quantitative comparisons among them. In general, correlation coefficient measures the relationship between two random variables X and Y. Correlation coefficient close to unity implies *good correlation*, or *linear association* between X and Y, while value near zero indicates little or no correlation. The definition of correlation coefficient is,

$$Correlation\ Coefficient = \frac{S_{xy}}{\sqrt{S_{xx}S_{yy}}}$$

$$S_{xx} = \sum_i (X_i - \bar{X})^2$$

$$S_{yy} = \sum_i (Y_i - \bar{Y})^2$$

$$S_{xy} = \sum_i (X_i - \bar{X})(Y_i - \bar{Y})$$

Complexity Measure	Project A Total errors	Project B Total errors	Project C Total errors	Over all Total errors
Zage design metric	0.61	0.66	0.73	0.57
Card design metric	0.58	0.61	0.74	0.50
Kafura design metric	0.57	0.60	0.85	0.50
Size (LOC)	0.61	0.69	0.85	0.60

Figure 11: Correlations of Design Complexity Measures with Errors

Correlations between different design complexity metrics and total weighted errors are shown in figure 11. Results indicate that Zage metric has better correlation with errors than the other two metrics. Correlations coefficients decreased when they are found on combined data of all projects. This indicates design complexity values of modules are more sensible for comparison within the project rather than across the projects. So instead of highlighting modules based on some fixed threshold values, it is better to highlight them using average and standard deviation values in a particular project.

Size has better correlation(0.60) with errors, than that of design metrics. But this is not much significant when compared to the correlation coefficient of zage design metric with errors(0.57).

4.3.3 Code metrics and Relationship to Errors

Similar to the design metrics, average values of source code complexity measures can also be used to highlight few modules in the system. Various related statistics when modules are highlighted based on the average values of source code metrics are given in figure 12. Results indicate that the effectiveness of cyclomatic complexity, size, volume, and programming effort are similar and better than the other source code metrics. Average values of difficulty, live and span size are unable to descriminate error prone modules from the others. Though cyclo, size, volume, and programming effort metrics are able to highlight many error prone modules, there

Characteristic	Project A	Project B	Project C
1. Modules with above average errors.	27	19	9
2. Modules with above average size.	34	23	14
3. Modules having above average size but with less than average errors.	8	6	7
4. Modules with above average cyclomatic number.	34	20	16
5. Modules having above average cyclomatic, but not error prone.	10	3	7
6. Modules with above average volume.	39	22	17
7. Modules having above average volume, but not error prone.	12	5	9
8. Modules with above average programming effort.	33	16	19
9. Modules with above average prog. effort, but not error prone.	11	1	3
10. Modules with above average difficulty.	46	24	11
11. Modules having above average difficulty, but not error prone.	20	11	5
12. Modules with above average live variables.	75	18	14
13. Modules with above average live variables, but not error prone	58	5	6
14. Modules with above average span size.	23	17	13
15. Modules with above average span size, but not error prone	16	4	5

Figure 12: Performance of Source Code Metrics

Complexity Measure	Project A Total errors	Project B Total errors	Project C Total errors	Over all Total errors
Size (LOC)	0.61	0.69	0.85	0.60
Cyclomatic Complexity	0.67	0.83	0.61	0.54
Live variables	0.37	0.60	0.58	0.20
Span size	0.40	0.58	0.51	0.18
Volume	0.73	0.78	0.65	0.70
Programming Effort	0.70	0.86	0.66	0.70
Difficulty	0.48	0.78	0.42	0.51

Figure 13: Correlations of Complexity Measures with Errors

exists few modules having these source code metrics above the average value, but not error prone.

Correlations between different source code complexity measures and total weighted errors are shown in the figure 13. Results in the tables indicates that,

- Cyclomatic complexity has significantly better correlation with errors than live variables, or average span size. It emphasises that programmer's perception of software complexity stems more from control flow complexity rather than data flow complexity.
- Module size, programming effort, and volume are equally good in predicting errors.
- "Difficulty", which is supposed to measure "ease of reading or writing" the source code, seems to be dubious. It does not have significant correlations with errors.

4.3.4 Relationship between different Metrics

Complexity Measure	Project A		Project B		Project C		Over all	
	Cyclo	Size	Cyclo	Size	Cyclo	Size	Cyclo	Size
Zage Design Metric	0.50	0.40	0.46	0.63	0.67	0.77	0.21	0.28
Card Design Metric	0.57	0.40	0.35	0.55	0.70	0.81	0.29	0.32
Kafura Design Metric	0.43	0.35	0.51	0.67	0.60	0.66	0.29	0.36
Size(LOC)	0.44	-	0.89	-	0.93	-	0.61	-

Figure 14: Correlations among Complexity Measures

Correlations coefficients between different metrics over the data of all projects are given in figure 14. Summary of these results are,

- No significant correlations between cyclomatic complexity values, and design metrics. Design metrics captures inter module complexity, where as Cyclomatic complexity captures intra module complexity. Inter module complexity need not imply intra module complexity. Correlation results indicate this fact. Hybrid approach will be more sensible, if software complexity is to be measured after the implementation phase.

- Similar to the above, module size also does not correlate significantly, with design complexity metrics. Again it is because size indicates intra module complexity, where as design complexity metrics indicate inter module complexity.

- There is a significant correlation between module size and cyclomatic complexity. It is because of the fact that both captures intra module complexity.

CHAPTER 5

Conclusions

Most widely used software development method *Water Fall Life Cycle Model* consists of various phases: Requirement Analysis, System Design, Detailed design, Implementation, and Testing. At the end of each phase one needs to evaluate the work carried out in that particular phase before continuing further. Generally, this is done by reviewing the work carried out in that phase. Reviewing the whole work in detail is cumbersome, and difficult. It not only wastes valuable resources, instead the real problematic areas of the software system may not get proper attention. We need some automated tools to highlight likely problematic areas in the system. This helps to catch the bugs in the system as soon as they are introduced. Cost of eliminating the bug will grow exponentially, as the time elapses. We have developed some tools, which can be applied at the end of design stage, or implementation stage.

Effectiveness of these tools have been checked with some projects developed by students. We used design complexity measurement tools to check the performance of various design metrics in predicting the error prone modules. We found that design metrics can be effectively used to predict error prone modules and also to compare the quality of two alternate designs. Similarly tools, which can be applied after the implementation phase are also experimented. Results of our experiments indicate that design complexity, size and control flow complexity are major contributors to

the errors made during development.

Complexity measurement tool will be of great use, when the size and complexity of the system is very large and is developed by many people. So, usefulness of tools developed in this work, need to be checked in large industrial environments. Once effectiveness of tools is validated across several projects developed in industrial environments, these software complexity measurement tools can be integrated into Automated Software Engineering Environment.

The major objective of the complexity measurement tools is to measure psychological perception of human being in developing a software. Tools developed, measure different aspects from which complexity arises. We found that some aspects are more important than the others. But if we can come up with a hybrid approach, which measures different aspects of the software, it would be more sensible. Some hybrid metrics are proposed in the literature, but they do not have empirical validation. This area needs some attention.

The psychological perception of complexity, relates to many software quality attributes. But there exists few quality attributes, which are not related to software complexity. For example, a software which is less complex may not be portable, or reuseable. So methods for measuring attributes like these are needed. No significant work is done in this area so far.

Bibliography

[1] Al-Janabi and E. Aspinwall. An evauation of software design using the demeter tool. *Software Engineering Journal*, November 1993.

[2] A.J. Albrecht and J.E. Gaffney. Software functions, source lines of code and developement effort pediction. *IEEE Transactions On Software Engineering*, November 1983.

[3] Benyon. and Tinker. Complexity measures in evaluating large systems. In *Proceedings of Workshop on quantitative Software models*, 1979.

[4] R.E. Berry and B.A.E. Meeking. A style analysis of c programs. *Communications of ACM*, January 1985.

[5] B.W. Boehm. *Software Engineering Economics.* Printice Hall, 1981.

[6] D. Card and R.L. Glass. *Measuring Software Design Quality.* Printice Hall, Englewood Cliffs, New Jersey, 1990.

[7] K. Christensen, G.P. Fitsos, and C.P. Smith. A perspective on software science. *IBM Systems Journal*, April 1981.

[8] T. DeMarco. *Controlling Software Projects.* Yourdon Press, NY, 1981.

[9] L.O. Ejiogu. A measure of software complexity. *SIGPLAN Notices*, March 1985.

[10] N.E. Fenton and A.A. Kapsoi. Metrics and software structure. *Journal of Information and Software Technology 29*, pages 301–320, July 1987.

[11] M. Halstead. *Elements of Software Science.* North-Holland, 1977.

[12] W. Hansen. Measurement of program complexity by the pair (cyclomatic number, opearator count). *ACM SIGPLAN Notices*, March 1978.

[13] S.M. Henry and D. Kafura. Software structure metrics based on information flow. *IEEE Transactions On software Engineering*, May 1981.

[14] H.F. Li and W.K. Chung. An empirical study of software metrics. *IEEE Transactions On software Engineering*, June 1987.

[15] T.J. McCabe. A complexity measure. *IEEE Transactions On software Engineering*, Dec 1977.

[16] G.J. Myer. *Reliable software through composite design.* Van Nostrand-Rheinhold, New York, 1979.

[17] E Oveido. Control flow, data flow and program complexity. *Proc COMPSAC*, 1980.

[18] R.G. Reynolds and E. Zannoni. *Encyclopedia of Software Engineering*, volume 1, chapter Metrics, pages 676–685. John Wiley & Sons, Inc, 1994.

[19] M.Z Wayne and M.Z Dolores. Evaluating design metrics on large scale software. *IEEE Software*, July 1993.

[20] M.D. Weiger. Program slicing. *Proc. Int. Conf. on Software Engineering*, 1981.

[21] S.S. Yau and S. Colofello. Some stability measures for software maintenance. *IEEE Transactions On software Engineering*, November 1980.

[22] B.H. Yin and J.W. Winchester. The establishment and use of measures to evaluate the quality of software designs. *Sigsoft Software Engg. notes*, 1978.

[23] Zuse. *Software Complexity.* Walter de Gruyter, Berlin, 1991.

APPENDIX A

Sample Input Program

```
/* Statement of the problem:
* This program counts the number of different words in a
* file. It outputs the each unique word and it's number of
* occurences in the file. Distiction is made between upper
* case and lower case letters.
*/
#include <stdio.h>
#include <string.h>
#define MAX_WORDS 50       /* Max. input words */
#define MAX_WORD_SIZE 50 /* Max. word size */

char *     WordList[MAX_WORDS];    /* list of words read from input file */
int        TotWords = 0;           /* Total words */
typedef struct                     /* info about word count */
{
           int WordIndex;          /* index into WordList */
           int count;              /* number of occurences of the word */
}CountRec;
CountRec WordCount[MAX_WORDS];
int        TotWordCount = 0;       /* Total unique words */
```

```
main(argc, argv)
int argc;
char * argv[];
{
        FILE *fd;
        if (argc ! = 2) /* excess or less arguments */
        {
                printf("usage: wordcount file \n");
                exit(-1);
        }
        /* if file does not exists */
        if ((fd = fopen(argv[1], "r")) == NULL )
        {
                printf("wordcount : can't open %s file \n", argv[1]);
                exit(-1);
        }
        /* read words from file and sort */
        GetSortedWordList(fd, WordList, &TotWords);
        if (TotWords > 0)
        {
                /* count the occurences of differrent words */
                CountWords(WordList, TotWords, WordCount, &TotWordCount);
                /* output the words and wordcounts */
                PrintCount(WordList, WordCount, TotWordCount);
        }
}

/* Read words words from file and sorts in ascending
* order
*/
```

```
GetSortedWordList(fd, WordList, TotWords)
FILE *fd;
char **WordList;
int *TotWords;
{
        /* read words from file */
        GetWordList(fd, WordList, TotWords);
        /* sort word list */
        SortWordList(WordList, (*TotWords));
}
/* Count the occurrences of each word
* Input WordList is sorted in ascending order
*/
CountWords(WordList, TotWords, WordCount, TotWordCount)
char ** WordList;
int TotWords;
CountRec * WordCount;
int * TotWordCount;
{
        char LastWord[MAX_WORD_SIZE];
        int index = 0;
        int i;
        /* initializations */
        strcpy(LastWord, WordList[0]);
        WordCount[index].count = 1;
        WordCount[index].WordIndex = 0;
        for(i = 1; i<TotWords; i++)
        {
                /* same as last word */
                if (strcmp(LastWord, WordList[i]) == 0)
```

```
                    (WordCount[index].count)++;
                else /* different word */
                {
                    strcpy(LastWord, WordList[i]);
                    WordCount[++index].WordIndex = i;
                    WordCount[index].count = 1;
                }
        }
        (*TotWordCount) = index + 1;
}

/* outputs the each word and it's occurrences in
* file on separate line
*/
PrintCount(WordList, WordCount, TotWordCount)
char **WordList;
CountRec *WordCount;
int TotWordCount;
{
        int i;
        printf("----------\n");
        printf("WORDS COUNT\n");
        printf("----------\n");
        for (i = 0; i<TotWordCount; i++)
            printf("%-25s %3d\n", WordList[WordCount[i].WordIndex],
                WordCount[i].count);
        printf("----------\n");
}
/* Read words from file
```

```
GetWordList(fd, WordList, TotWords)
FILE * fd;
char ** WordList;
int * TotWords;
{       char word[MAX_WORD_SIZE];
        int i = 0;
        while (fscanf(fd,"%s", word) != EOF)
        {
                WordList[i] = (char *) malloc(strlen(word)+1);
                strcpy(WordList[i], word);
                i++;
        }
        (*TotWords) = i;
}
/* sort the words in WordList in ascending order
*/
SortWordList(WordList, TotWords)
char ** WordList;
int TotWords;
{
        int i, j;
        char *TempChar;
        for(i = 0; i < TotWords; i++)
                for(j = i+1; j < TotWords; j++)
                        if (strcmp(WordList[i], WordList[j]) > 0)
                        {
                                /* interchange */
                                TempChar = WordList[i];
                                WordList[i]= WordList[j];
                                WordList[j] = TempChar;
```

```
GetWordList(fd, WordList, TotWords)
FILE * fd;
char ** WordList;
int * TotWords;
{       char word[MAX_WORD_SIZE];
        int i = 0;
        while (fscanf(fd,"%s", word) != EOF)
        {
                WordList[i] = (char *) malloc(strlen(word)+1);
                strcpy(WordList[i], word);
                i++;
        }
        (*TotWords) = i;
}
/* sort the words in WordList in ascending order
*/
SortWordList(WordList, TotWords)
char ** WordList;
int TotWords;
{
        int i, j;
        char *TempChar;
        for(i = 0; i < TotWords; i++)
                for(j = i+1; j < TotWords; j++)
                        if (strcmp(WordList[i], WordList[j]) > 0)
                        {
                                /* interchange */
                                TempChar = WordList[i];
                                WordList[i]= WordList[j];
                                WordList[j] = TempChar;
```

```
                    }
}
```

APPENDIX B

Sample Program Design Specification

```
/* Statement of the problem:
* This program counts the number of different words in a
* file. It outputs the each unique word and it's number of
* occurences in the file. Distiction is made between upper
* case and lower case letters.
*/
char        *WordList[MAX_WORDS]; /* list of words read from input file */
int         TotWords = 0;                 /* Total words */
typedef struct        /* info about word count */
{
            int WordIndex;          /* index into WordList */
            int count;              /* number of occurences of the word */
}CountRec;
CountRec WordCount[MAX_WORDS];
int TotWordCount = 0;           /* Total unique words */
main(IN: argc,argv)
int argc;
char *argv[];
```

```
{
        SUBORDINATES:   GetSortedWordList(),
                        CountWords(),
                        PrintCount();
        SIZE: 25
}
/* Read words words from file and sorts in ascending
* order
*/
GetSortedWordList(IN: fd, OUT: WordList,TotWords)
FILE *fd;
char **WordList;
int *TotWords;
{
        SUBORDINATES:   GetWordList(),
                        SortWordList();
        SIZE: 10
}
/* Count the occurrences of each word
* Input WordList is sorted in ascending oreder
*/
CountWords(IN: WordList,TotWords,OUT: WordCount,TotWordCount)
char **WordList;
int TotWords;
CountRec *WordCount;
int *TotWordCount;
{
        SUBORDINATES:
        SIZE: 30
}
```

```
/* outputs the each word and it's occurrences in
* file on separate line
*/
PrintCount(IN: WordList,WordCount,TotWordCount)
char **WordList;
CountRec *WordCount;
int TotWordCount;
{
        SUBORDINATES:
        SIZE: 15
}
/* Read words from file
*/
GetWordList(IN: fd,OUT: WordList,TotWords)
FILE * fd;
char ** WordList;
int * TotWords;
{
        SUBORDINATES:
        SIZE: 15
}
/* sort the words in WordList in ascending order
*/
SortWordList(IN_OUT: WordList,TotWords)
char ** WordList;
int TotWords;
{
        SUBORDINATES:
        SIZE: 15
}
```

APPENDIX C

Sample Outputs

Output produced by dmetric for the program design specifications listed in Appendix.A. dmetre also produces the same output when it is applied on it's program listings in Appendix B. Here *zage metric* is considered as design metric formula, and Method B is used in deciding the module as *error prone*, or *complex*.

OVERALL METRICS

#modules: 6	Total size:	108	Average size:	18	Std.Deviation: 7
Total design	complexity:	72	Avg. complexity:	12	Std.Deviation: 15

Deviation of the structure chart from a tree(with out considering leaves) = 0

ERROR PRONE MODULES

COMPLEX MODULES

0) main

call_in: 1 call_out: 3 inflow: 6 outflow: 6 size:25

design complexity: 39

2) CountWords

call_in: 1 call_out: 0 inflow: 2 outflow: 2 size:30

design complexity: 4

DETAILED LISTING

COMPLEXITY OF MODULES

0) main

call_in: 1 call_out: 3 inflow: 6 outflow: 6 size:25

design complexity: 39 diagnosis:Complex

1) GetSortedWordList

call_in: 1 call_out: 2 inflow: 5 outflow: 5 size:10

design complexity: 27 diagnosis:ok

2) CountWords

call_in: 1 call_out: 0 inflow: 2 outflow: 2 size:30

design complexity: 4 diagnosis:Complex

3) PrintCount

call_in: 1 call_out: 0 inflow: 3 outflow: 0 size:15

design complexity: 0 diagnosis:ok

4) GetWordList

call_in: 1 call_out: 0 inflow: 1 outflow: 2 size:15

design complexity: 2 diagnosis:ok

5) SortWordList

call_in: 1 call_out: 0 inflow: 2 outflow: 2 size:15

design complexity: 4 diagnosis:ok

Cyclomatic Complexity values and Halstead Measures for the sample program listed in Appendix B.

Module	size	Volume	Ease of Reading or writing	Programming effort
main	26	414	42	17
CountWords	30	394	36	14
SortWordList	18	254	46	11
GetWordList	16	235	36	8
PrintCount	15	214	24	5
GetSortedWordList	10	84	25	2

Module	Size	Cyclomatic complexity
SortWordList	18	4
CountWords	30	3
main	26	3
GetWordList	16	2
PrintCount	15	2
GetSortedWordList	10	1

#modules	:	6			
Total Size	:	115	Avg. Size	:	19
Total Volume	:	1595	Avg. Volme	:	265
Total Cyclomatic	:	15	Avg. Cyclomatic	:	2

Ouput produced by **live.**

DATA FLOW COMPLEXITY OF SUBROUTINES

Module	Avg. number of live variables per stmt	Avg. span of variables
CountWords	4.27	1.54
SortWordList	3.17	0.24
GetWordList	2.60	0.00
GetSortedWordList	2.50	0.00
main	1.89	0.33
PrintCount	1.25	0.00

Style Measures computed by **style,**

Score for the Programming Style : 63/100 **** **GOOD**

STYLE METRICS FOR THE WHOLE PROGRAM

Programming Style Feature	Metric	Diagnosis
Avg. size of the module	19	ACCEPTABLE
Avg. identifier length	7	ACCEPTABLE
% of comment lines	20	ACCEPTABLE
% of blank lines	14	LESS
Avg. number of chars per line	10	LESS
Avg. number of embeded spaces per line	0	VERY LESS
Total no. of reserved words used	17	ACCEPTABLE
Number of files included	2	LESS
Number of goto's used	0	